16 – Networking

When it comes to networking, there is probably nothing that cannot be done with Linux. Linux is used to build all sorts of networking systems and appliances, including firewalls, routers, name servers, network-attached storage (NAS) boxes and on and on.

Just as the subject of networking is vast, so are the number of commands that can be used to configure and control it. We will focus our attention on just a few of the most frequently used ones. The commands chosen for examination include those used to monitor networks and those used to transfer files. In addition, we are going to explore the `ssh` program that is used to perform remote logins. This chapter will cover the following commands:

- `ping` – Send an ICMP ECHO_REQUEST to network hosts
- `traceroute` – Print the route packets trace to a network host
- `ip` – Show / manipulate routing, devices, policy routing and tunnels
- `netstat` – Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships
- `ftp` – Internet file transfer program
- `wget` – Non-interactive network downloader
- `ssh` – OpenSSH SSH client (remote login program)

We're going to assume a little background in networking. In this, the Internet age, everyone using a computer needs a basic understanding of networking concepts. To make full use of this chapter we should be familiar with the following terms:

- Internet protocol (IP) address
- Host and domain name
- Uniform resource identifier (URI)

Please see the "Further Reading" section below for some useful articles regarding these terms.

> **Note:** Some of the commands we will cover may (depending on your distribution) require the installation of additional packages from your distribution's repositories, and some may require superuser privileges to execute.

Examining and Monitoring a Network

Even if you're not the system administrator, it's often helpful to examine the performance and operation of a network.

ping

The most basic network command is `ping`. The `ping` command sends a special network packet called an ICMP ECHO_REQUEST to a specified host. Most network devices receiving this packet will reply to it, allowing the network connection to be verified.

> **Note:** It is possible to configure most network devices (including Linux hosts) to ignore these packets. This is usually done for security reasons, to partially obscure a host from a potential attacker. It is also common for firewalls to be configured to block ICMP traffic.

For example, to see whether we can reach `linuxcommand.org` (one of our favorite sites ;-), we can use `ping` like this:

```
[me@linuxbox ~]$ ping linuxcommand.org
```

Once started, `ping` continues to send packets at a specified interval (default is one second) until it is interrupted.

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=1
ttl=43 time=107 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=2
ttl=43 time=108 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=3
ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=4
ttl=43 time=106 ms
```

```
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=5
ttl=43 time=105 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=6
ttl=43 time=107 ms

--- linuxcommand.org ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 6010ms
rtt min/avg/max/mdev = 105.647/107.052/108.118/0.824 ms
```

After it is interrupted (in this case after the sixth packet) by pressing `Ctrl-c`, `ping` prints performance statistics. A properly performing network will exhibit 0 percent packet loss. A successful "ping" will indicate that the elements of the network (its interface cards, cabling, routing, and gateways) are in generally good working order.

traceroute

The `traceroute` program (some systems use the similar `tracepath` program instead) lists all the "hops" network traffic takes to get from the local system to a specified host. For example, to see the route taken to reach `slashdot.org`, we would do this:

```
[me@linuxbox ~]$ traceroute slashdot.org
```

The output looks like this:

```
traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte
packets
 1  ipcop.localdomain (192.168.1.1)  1.066 ms  1.366 ms  1.720 ms
 2  * * *
 3  ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9)  14.622
ms  14.885 ms  15.169 ms
 4  po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154)  17.634
ms  17.626 ms  17.899 ms
 5  po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158)  15.992
ms  15.983 ms  16.256 ms
 6  po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5)  22.835
ms  14.233 ms  14.405 ms
 7  po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34)  16.154
ms  13.600 ms  18.867 ms
 8  te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77)
21.951 ms  21.073 ms  21.557 ms
```

```
 9  pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10)
22.917 ms  21.884 ms  22.126 ms
10  204.70.144.1 (204.70.144.1)  43.110 ms  21.248 ms  21.264 ms
11  cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93)  21.857 ms
cr2-pos-0-0-3-1.newyork.savvis.net (204.70.204.238)  19.556 ms cr1-
pos-0-7-3-1.newyork.savvis.net (204.70.195.93)  19.634 ms
12  cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109)  41.586 ms
42.843 ms cr2-tengig-0-0-2-0.chicago.savvis.net (204.70.196.242)
43.115 ms
13  hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net
(204.70.195.122)  44.215 ms  41.833 ms  45.658 ms
14  csr1-ve241.elkgrovech3.savvis.net (216.64.194.42)  46.840 ms
43.372 ms  47.041 ms
15  64.27.160.194 (64.27.160.194)  56.137 ms  55.887 ms  52.810 ms
16  slashdot.org (216.34.181.45)  42.727 ms  42.016 ms  41.437 ms
```

In the output, we can see that connecting from our test system to `slashdot.org` requires traversing 16 routers. For routers that provided identifying information, we see their hostnames, IP addresses, and performance data, which includes three samples of round-trip time from the local system to the router. For routers that do not provide identifying information (because of router configuration, network congestion, firewalls, etc.), we see asterisks as in the line for hop number 2. In cases where routing information is blocked, we can sometimes overcome this by adding either the `-T` or `-I` option to the `traceroute` command.

ip

The `ip` program is a multi-purpose network configuration tool that makes use of the full range networking of features available in modern Linux kernels. It replaces the earlier and now deprecated `ifconfig` program. With `ip`, we can examine a system's network interfaces and routing table.

```
[me@linuxbox ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
```

```
state UP group default qlen 1000
    link/ether ac:22:0b:52:cf:84 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.14/24 brd 192.168.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::ae22:bff:fe52:cf84/64 scope link
       valid_lft forever preferred_lft forever
```

In the example above, we see that our test system has two network interfaces. The first, called `lo`, is the *loopback interface*, a virtual interface that the system uses to "talk to itself" and the second, called `eth0`, is the Ethernet interface.

When performing casual network diagnostics, the important things to look for are the presence of the word `UP` in the first line for each interface, indicating that the network interface is enabled, and the presence of a valid IP address in the `inet` field on the third line. For systems using Dynamic Host Configuration Protocol (DHCP), a valid IP address in this field will verify that the DHCP is working.

`netstat`

The `netstat` program is used to examine various network settings and statistics. Through the use of its many options, we can look at a variety of features in our network setup. Using the `-ie` option, we can examine the network interfaces in our system.

```
[me@linuxbox ~]$ netstat -ie
eth0    Link encap:Ethernet  HWaddr 00:1d:09:9b:99:67
        inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
        inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
        TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        RX bytes:153098921 (146.0 MB)  TX bytes:261035246 (248.9 MB)
        Memory:fdfc0000-fdfe0000

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:2208 errors:0 dropped:0 overruns:0 frame:0
        TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:111490 (108.8 KB)  TX bytes:111490 (108.8 KB)
```

Using the `-r` option will display the kernel's network routing table. This shows how the network is configured to send packets from network to network.

```
[me@linuxbox ~]$ netstat -r
Kernel IP routing table
Destination  Gateway     Genmask       Flags   MSS Window  irtt Iface

192.168.1.0  *           255.255.255.0 U         0 0          0 eth0
default      192.168.1.1 0.0.0.0       UG        0 0          0 eth0
```

In this simple example, we see a typical routing table for a client machine on a local area network (LAN) behind a firewall/router. The first line of the listing shows the destination `192.168.1.0`. IP addresses that end in zero refer to networks rather than individual hosts, so this destination means any host on the LAN. The next field, `Gateway`, is the name or IP address of the gateway (router) used to go from the current host to the destination network. An asterisk in this field indicates that no gateway is needed.

The last line contains the destination `default`. This means any traffic destined for a network that is not otherwise listed in the table. In our example, we see that the gateway is defined as a router with the address of `192.168.1.1`, which presumably knows what to do with the destination traffic.

Like `ip`, the `netstat` program has many options and we have looked only at a couple. Check out the `ip` and `netstat` man pages for a complete list.

Transporting Files Over a Network

What good is a network unless we can move files across it? There are many programs that move data over networks. We will cover two of them now and several more in later sections.

`ftp`

One of the true "classic" programs, `ftp` gets its name from the protocol it uses, the *File Transfer Protocol*. FTP was once the most widely used method of downloading files over the Internet. Most, if not all, web browsers support it, and you often see URIs starting with the protocol `ftp://`.

Before there were web browsers, there was the `ftp` program. `ftp` is used to communicate with *FTP servers*, machines that contain files that can be uploaded and downloaded over a network.

FTP (in its original form) is not secure because it sends account names and passwords in

cleartext. This means they are not encrypted and anyone *sniffing* the network can see them. Because of this, almost all FTP done over the Internet is done by *anonymous FTP servers*. An anonymous server allows anyone to log in using the login name "anonymous" and a meaningless password.

In the example below, we show a typical session with the `ftp` program downloading an Ubuntu iso image located in the `/pub/cd_images/Ubuntu-18.04` directory of the anonymous FTP server `fileserver`:

```
[me@linuxbox ~]$ ftp fileserver
Connected to fileserver.localdomain.
220 (vsFTPd 2.0.1)
Name (fileserver:me): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/cd_images/Ubuntu-18.04
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-rw-r--    1 500      500      733079552 Apr 25 03:53 ubuntu-
18.04-desktop-amd64.iso
226 Directory send OK.
ftp> lcd Desktop
Local directory now /home/me/Desktop
ftp> get ubuntu-18.04-desktop-amd64.iso
local: ubuntu-18.04-desktop-amd64.iso remote: ubuntu-18.04-desktop-
amd64.iso
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for ubuntu-18.04-desktop-
amd64.iso (733079552 bytes).
226 File send OK.
733079552 bytes received in 68.56 secs (10441.5 kB/s)
ftp> bye
```

Table 16-1 provides an explanation of the commands entered during this session.

Table 16-1: Examples of Interactive `ftp` Commands

Command	Meaning

`ftp fileserver`	Invoke the `ftp` program and have it connect to the FTP server `fileserver`.
`anonymous`	Login name. After the login prompt, a password prompt will appear. Some servers will accept a blank password; others will require a password in the form of an email address. In that case, try something like `user@example.com`.
`cd pub/cd_images/Ubuntu-18.04`	Change to the directory on the remote system containing the desired file. Note that on most anonymous FTP servers, the files for public downloading are found somewhere under the `pub` directory.
`ls`	List the directory on the remote system.
`lcd Desktop`	Change the directory on the local system to `~/Desktop`. In the example, the `ftp` program was invoked when the working directory was `~`. This command changes the working directory to `~/Desktop`.
`get ubuntu-18.04-desktop-amd64.iso`	Tell the remote system to transfer the file `ubuntu-18.04-desktop-amd64.iso` to the local system. Since the working directory on the local system was changed to `~/Desktop`, the file will be downloaded there.
`bye`	Log off the remote server and end the `ftp` program session. The commands `quit` and `exit` may also be used.

Typing `help` at the `ftp>` prompt will display a list of the supported commands. Using `ftp` on a server where sufficient permissions have been granted, it is possible to perform

many ordinary file management tasks. It's clumsy, but it does work.

lftp – A Better ftp

ftp is not the only command-line FTP client. In fact, there are many. One of the better (and more popular) ones is lftp by Alexander Lukyanov. It works much like the traditional ftp program but has many additional convenience features including multiple-protocol support (including HTTP), automatic retry on failed downloads, background processes, tab completion of path names, and many more.

wget

Another popular command-line program for file downloading is wget. It is useful for downloading content from both web and FTP sites. Single files, multiple files, and even entire sites can be downloaded. To download the first page of linuxcommand.org we could do this:

```
[me@linuxbox ~]$ wget http://linuxcommand.org/index.php
--11:02:51--  http://linuxcommand.org/index.php
           => `index.php'
Resolving linuxcommand.org... 66.35.250.210
Connecting to linuxcommand.org|66.35.250.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]

    [ <=>                                 ] 3,120         --.--K/s

11:02:51 (161.75 MB/s) - `index.php' saved [3120]
```

The program's many options allow wget to recursively download, download files in the background (allowing you to log off but continue downloading), and complete the download of a partially downloaded file. These features are well documented in its better-than-average man page.

Secure Communication with Remote Hosts

For many years, Unix-like operating systems have had the ability to be administered remotely via a network. In the early days, before the general adoption of the Internet, there were a couple of popular programs used to log in to remote hosts. These were the rlogin and telnet programs. These programs, however, suffer from the same fatal flaw that the ftp program does; they transmit all their communications (including login

names and passwords) in cleartext. This makes them wholly inappropriate for use in the Internet age.

ssh

To address this problem, a new protocol called Secure Shell (SSH) was developed. SSH solves the two basic problems of secure communication with a remote host.

1. It authenticates that the remote host is who it says it is (thus preventing so-called man-in-the-middle attacks).
2. It encrypts all of the communications between the local and remote hosts.

SSH consists of two parts. An SSH server runs on the remote host, listening for incoming connections, by default, on port 22, while an SSH client is used on the local system to communicate with the remote server.

Most Linux distributions ship an implementation of SSH called OpenSSH from the OpenBSD project. Some distributions include both the client and the server packages by default (for example, Red Hat), while others (such as Ubuntu) only supply the client. To enable a system to receive remote connections, it must have the `OpenSSH-server` package installed, configured and running, and (if the system either is running or is behind a firewall) it must allow incoming network connections on TCP port 22.

Tip: If you don't have a remote system to connect to but want to try these examples, make sure the `OpenSSH-server` package is installed on your system and use `localhost` as the name of the remote host. That way, your machine will create network connections with itself.

The SSH client program used to connect to remote SSH servers is called, appropriately enough, `ssh`. To connect to a remote host named `remote-sys`, we would use the `ssh` client program like so:

```
[me@linuxbox ~]$ ssh remote-sys
The authenticity of host 'remote-sys (192.168.1.4)' can't be
established.
RSA key fingerprint is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Are you sure you want to continue connecting (yes/no)?
```

The first time the connection is attempted, a message is displayed indicating that the authenticity of the remote host cannot be established. This is because the client program has never seen this remote host before. To accept the credentials of the remote host, enter

"yes" when prompted. Once the connection is established, the user is prompted for a password:

```
Warning: Permanently added 'remote-sys,192.168.1.4' (RSA) to the list
of known hosts.
me@remote-sys's password:
```

After the password is successfully entered, we receive the shell prompt from the remote system.

```
Last login: Sat Aug 30 13:00:48 2016
[me@remote-sys ~]$
```

The remote shell session continues until the user enters the `exit` command at the remote shell prompt, thereby closing the remote connection. At this point, the local shell session resumes, and the local shell prompt reappears.

It is also possible to connect to remote systems using a different username. For example, if the local user "me" had an account named "bob" on a remote system, user `me` could log in to the account `bob` on the remote system as follows:

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys's password:
Last login: Sat Aug 30 13:03:21 2016
[bob@remote-sys ~]$
```

As stated earlier, `ssh` verifies the authenticity of the remote host. If the remote host does not successfully authenticate, the following message appears:

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle
attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
```

```
Please contact your system administrator.
Add correct host key in /home/me/.ssh/known_hosts to get rid of this
message.
Offending key in /home/me/.ssh/known_hosts:1
RSA host key for remote-sys has changed and you have requested strict
checking.
Host key verification failed.
```

This message is caused by one of two possible situations. First, an attacker may be attempting a man-in-the-middle attack. This is rare, since everybody knows that `ssh` alerts the user to this. The more likely culprit is that the remote system has been changed somehow; for example, its operating system or SSH server has been reinstalled. In the interests of security and safety, however, the first possibility should not be dismissed out of hand. Always check with the administrator of the remote system when this message occurs.

After it has been determined that the message is because of a benign cause, it is safe to correct the problem on the client side. This is done by using a text editor (`vim` perhaps) to remove the obsolete key from the `~/.ssh/known_hosts` file. In the example message above, we see this:

```
Offending key in /home/me/.ssh/known_hosts:1
```

This means that the first line of the `known_hosts` file contains the offending key. Delete this line from the file, and the `ssh` program will be able to accept new authentication credentials from the remote system.

Besides opening a shell session on a remote system, `ssh` allows us to execute a single command on a remote system. For example, to execute the `free` command on a remote host named `remote-sys` and have the results displayed on the local system, use this:

```
[me@linuxbox ~]$ ssh remote-sys free
me@twin4's password:
             total       used       free     shared    buffers     cached

Mem:        775536     507184     268352          0     110068     154596

-/+ buffers/cache:     242520     533016
Swap:      1572856          0    1572856
[me@linuxbox ~]$
```

It’s possible to use this technique in more interesting ways, such as the following exam-

ple in which we perform an ls on the remote system and redirect the output to a file on the local system:

```
[me@linuxbox ~]$ ssh remote-sys 'ls *' > dirlist.txt
me@twin4's password:
[me@linuxbox ~]$
```

Notice the use of the single quotes in the command above. This is done because we do not want the pathname expansion performed on the local machine; rather, we want it to be performed on the remote system. Likewise, if we had wanted the output redirected to a file on the remote machine, we could have placed the redirection operator and the filename within the single quotes.

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

Tunneling with SSH

Part of what happens when you establish a connection with a remote host via SSH is that an *encrypted tunnel* is created between the local and remote systems. Normally, this tunnel is used to allow commands typed at the local system to be transmitted safely to the remote system and for the results to be transmitted safely back. In addition to this basic function, the SSH protocol allows most types of network traffic to be sent through the encrypted tunnel, creating a sort of virtual private network (VPN) between the local and remote systems.

Perhaps the most common use of this feature is to allow X Window system traffic to be transmitted. On a system running an X server (that is, a machine displaying a GUI), it is possible to launch and run an X client program (a graphical application) on a remote system and have its display appear on the local system. It's easy to do; here's an example. Let's say we are sitting at a Linux system called `linuxbox` that is running an X server, and we want to run the `xload` program on a remote system named `remote-sys` to see the program's graphical output on our local system. We could do this:

```
[me@linuxbox ~]$ ssh -X remote-sys
me@remote-sys's password:
Last login: Mon Sep 08 13:23:11 2016
[me@remote-sys ~]$ xload
```

After the `xload` command is executed on the remote system, its window appears on the local system. On some systems, you may need to use the "-Y" option rather than the "-X" option to do this.

`scp` and `sftp`

The OpenSSH package also includes two programs that can make use of an SSH-encrypted tunnel to copy files across the network. The first, `scp` (secure copy) is used much like the familiar `cp` program to copy files. The most notable difference is that the source or destination pathnames may be preceded with the name of a remote host, followed by a colon character. For example, if we wanted to copy a document named `document.txt` from our home directory on the remote system, `remote-sys`, to the current working directory on our local system, we could do this:

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys's password:
document.txt                              100% 5581     5.5KB/s   00:00
[me@linuxbox ~]$
```

As with `ssh`, you may apply a username to the beginning of the remote host's name if the desired remote host account name does not match that of the local system.

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

The second SSH file-copying program is `sftp` which, as its name implies, is a secure replacement for the `ftp` program. `sftp` works much like the original `ftp` program that we used earlier; however, instead of transmitting everything in cleartext, it uses an SSH encrypted tunnel. `sftp` has an important advantage over conventional `ftp` in that it does not require an FTP server to be running on the remote host. It requires only the SSH server. This means that any remote machine that can connect with the SSH client can also be used as an FTP-like server. Here is a sample session:

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
```

```
ubuntu-8.04-desktop-i386.iso
sftp> lcd Desktop
sftp> get ubuntu-8.04-desktop-i386.iso
Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-
desktop-i386.iso
/home/me/ubuntu-8.04-desktop-i386.iso 100%  699MB   7.4MB/s   01:35
sftp> bye
```

Tip: The SFTP protocol is supported by many of the graphical file managers found in Linux distributions. Using either GNOME or KDE, we can enter a URI beginning with `sftp://` into the location bar and operate on files stored on a remote system running an SSH server.

An SSH Client for Windows?

Let's say you are sitting at a Windows machine but you need to log in to your Linux server and get some real work done; what do you do? Get an SSH client program for your Windows box, of course! There are a number of these. The most popular one is probably PuTTY by Simon Tatham and his team. The PuTTY program displays a terminal window and allows a Windows user to open an SSH (or telnet) session on a remote host. The program also provides analogs for the `scp` and `sftp` programs.

PuTTY is available at http://www.chiark.greenend.org.uk/~sgtatham/putty/

Summing Up

In this chapter, we surveyed the field of networking tools found on most Linux systems. Since Linux is so widely used in servers and networking appliances, there are many more that can be added by installing additional software. But even with the basic set of tools, it is possible to perform many useful network-related tasks.

Further Reading

- For a broad (albeit dated) look at network administration, the Linux Documentation Project provides the *Linux Network Administrator's Guide*: http://tldp.org/LDP/nag2/index.html

- Wikipedia contains many good networking articles. Here are some of the basics:
  http://en.wikipedia.org/wiki/Internet_protocol_address
  http://en.wikipedia.org/wiki/Host_name
  http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

17 – Searching for Files

As we have wandered around our Linux system, one thing has become abundantly clear: a typical Linux system has a lot of files! This begs the question, "How do we find things?" We already know that the Linux file system is well organized according to conventions passed down from one generation of Unix-like systems to the next, but the sheer number of files can present a daunting problem.

In this chapter, we will look at two tools that are used to find files on a system.

- `locate` – Find files by name
- `find` – Search for files in a directory hierarchy

We will also look at a command that is often used with file-search commands to process the resulting list of files.

- `xargs` – Build and execute command lines from standard input

In addition, we will introduce a couple of commands to assist us in our explorations.

- `touch` – Change file times
- `stat` – Display file or file system status

`locate` – Find Files the Easy Way

The `locate` program performs a rapid database search of pathnames, and then outputs every name that matches a given substring. Say, for example, we want to find all the programs with names that begin with `zip`. Since we are looking for programs, we can assume that the name of the directory containing the programs would end with `bin/`. Therefore, we could try to use `locate` this way to find our files:

```
[me@linuxbox ~]$ locate bin/zip
```

`locate` will search its database of pathnames and output any that contain the string `bin/zip`.

```
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

If the search requirement is not so simple, we can combine `locate` with other tools such as `grep` to design more interesting searches.

```
[me@linuxbox ~]$ locate zip | grep bin
/bin/bunzip2
/bin/bzip2
/bin/bzip2recover
/bin/gunzip
/bin/gzip
/usr/bin/funzip
/usr/bin/gpg-zip
/usr/bin/preunzip
/usr/bin/prezip
/usr/bin/prezip-bin
/usr/bin/unzip
/usr/bin/unzipsfx
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

The `locate` program has been around for a number of years, and there are several variants in common use. The two most common ones found in modern Linux distributions are `slocate` and `mlocate`, though they are usually accessed by a symbolic link named `locate`. The different versions of `locate` have overlapping options sets. Some versions include regular expression matching (which we'll cover in Chapter 19, "Regular Expressions") and wildcard support. Check the man page for `locate` to determine which version of `locate` is installed.

Where Does the `locate` Database Come From?

You may notice that, on some distributions, `locate` fails to work just after the system is installed, but if you try again the next day, it works fine. What gives? The `locate` database is created by another program named `updatedb`. Usually, it is run periodically as a *cron job*, that is, a task performed at regular intervals by the cron daemon. Most systems equipped with `locate` run `updatedb` once a day. Since the database is not updated continuously, you will notice that very recent files do not show up when using `locate`. To overcome this, it's possible to run the `updatedb` program manually by becoming the superuser and running `updatedb` at the prompt.

`find` – Find Files the Hard Way

While the `locate` program can find a file based solely on its name, the `find` program searches a given directory (and its subdirectories) for files based on a variety of attributes. We're going to spend a lot of time with `find` because it has a lot of interesting features that we will see again and again when we start to cover programming concepts in later chapters.

In its simplest use, `find` is given one or more names of directories to search. For example, to produce a listing of our home directory we can use this:

```
[me@linuxbox ~]$ find ~
```

On most active user accounts, this will produce a large list. Since the list is sent to standard output, we can pipe the list into other programs. Let's use `wc` to count the number of files.

```
[me@linuxbox ~]$ find ~ | wc -l
47068
```

Wow, we've been busy! The beauty of `find` is that it can be used to identify files that meet specific criteria. It does this through the (slightly strange) application of *options*, *tests*, and *actions*. We'll look at the tests first.

Tests

Let's say we want a list of directories from our search. To do this, we could add the following test:

```
[me@linuxbox ~]$ find ~ -type d | wc -l
1695
```

Adding the test `-type d` limited the search to directories. Conversely, we could have limited the search to regular files with this test:

```
[me@linuxbox ~]$ find ~ -type f | wc -l
38737
```

Table 17-1 lists the common file type tests supported by find.

Table 17-1: `find` File Types

File Type	Description
`b`	Block special device file
`c`	Character special device file
`d`	Directory
`f`	Regular file
`l`	Symbolic link

We can also search by file size and filename by adding some additional tests. Let's look for all the regular files that match the wildcard pattern `*.JPG` and are larger than one megabyte.

```
[me@linuxbox ~]$ find ~ -type f -name "*.JPG" -size +1M | wc -l
840
```

In this example, we add the `-name` test followed by the wildcard pattern. Notice how we enclose it in quotes to prevent pathname expansion by the shell. Next, we add the `-size` test followed by the string "+1M". The leading plus sign indicates that we are looking for files larger than the specified number. A leading minus sign would change the meaning of the string to be smaller than the specified number. Using no sign means, "match the value

exactly." The trailing letter `M` indicates that the unit of measurement is megabytes. Table 17-2 lists the characters that can be used to specify units.

Table 17-2: `find` Size Units

Character	Unit
`b`	512-byte blocks. This is the default if no unit is specified.
`c`	Bytes.
`w`	2-byte words.
`k`	Kilobytes (units of 1024 bytes).
`M`	Megabytes (units of 1048576 bytes).
`G`	Gigabytes (units of 1073741824 bytes).

`find` supports a large number of tests. Table 17-3 provides a rundown of the common ones. Note that in cases where a numeric argument is required, the same `+` and `-` notation discussed above can be applied.

Table 17-3: `find` Tests

Test	Description
`-cmin` *`n`*	Match files or directories whose content or attributes were last modified exactly *`n`* minutes ago. To specify less than *`n`* minutes ago, use `-`*`n`*, and to specify more than *`n`* minutes ago, use `+`*`n`*.
`-cnewer` *`file`*	Match files or directories whose contents or attributes were last modified more recently than those of *`file`*.
`-ctime` *`n`*	Match files or directories whose contents or attributes were last modified *`n`**24 hours ago.
`-empty`	Match empty files and directories.
`-group` *`name`*	Match file or directories belonging to *`group`*. *`group`* may be expressed either as a group name or as a numeric group ID.
`-iname` *`pattern`*	Like the `-name` test but case-insensitive.
`-inum` *`n`*	Match files with inode number *`n`*. This is helpful for finding all the hard links to a particular inode.

`-mmin` *`n`*	Match files or directories whose contents were last modified *`n`* minutes ago.
`-mtime` *`n`*	Match files or directories whose contents were last modified *`n`**24 hours ago.
`-name` *`pattern`*	Match files and directories with the specified wildcard *`pattern`*.
`-newer` *`file`*	Match files and directories whose contents were modified more recently than the specified *`file`*. This is useful when writing shell scripts that perform file backups. Each time you make a backup, update a file (such as a log) and then use `find` to determine which files have changed since the last update.
`-nouser`	Match file and directories that do not belong to a valid user. This can be used to find files belonging to deleted accounts or to detect activity by attackers.
`-nogroup`	Match files and directories that do not belong to a valid group.
`-perm` *`mode`*	Match files or directories that have permissions set to the specified *`mode`*. *`mode`* can be expressed by either octal or symbolic notation.
`-samefile` *`name`*	Similar to the `-inum` test. Match files that share the same inode number as file *`name`*.
`-size` *`n`*	Match files of size *`n`*.
`-type` *`c`*	Match files of type *`c`*.
`-user` *`name`*	Match files or directories belonging to user *`name`*. The user may be expressed by a username or by a numeric user ID.

This is not a complete list. The `find` man page has all the details.

Operators

Even with all the tests that `find` provides, we may still need a better way to describe the *logical relationships* between the tests. For example, what if we needed to determine whether all the files and subdirectories in a directory had secure permissions? We would look for all the files with permissions that are not 0600 and the directories with permissions that are not 0700. Fortunately, `find` provides a way to combine tests using *logical*

operators to create more complex logical relationships. To express the aforementioned test, we could do this:

```
[me@linuxbox ~]$ find ~ \( -type f -not -perm 0600 \) -or \( -type d
-not -perm 0700 \)
```

Yikes! That sure looks weird. What is all this stuff? Actually, the operators are not that complicated once you get to know them. Table 17-4 describes the logical operators used with `find`.

Table 17-4: `find` Logical Operators

Operator	Description
`-and`	Match if the tests on both sides of the operator are true. This can be shortened to `-a`. Note that when no operator is present, `-and` is implied by default.
`-or`	Match if a test on either side of the operator is true. This can be shortened to `-o`.
`-not`	Match if the test following the operator is false. This can be abbreviated with an exclamation point (`!`).
`( )`	Groups tests and operators together to form larger expressions. This is used to control the precedence of the logical evaluations. By default, `find` evaluates from left to right. It is often necessary to override the default evaluation order to obtain the desired result. Even if not needed, it is helpful sometimes to include the grouping characters to improve the readability of the command. Note that since the parentheses have special meaning to the shell, they must be quoted when using them on the command line to allow them to be passed as arguments to `find`. Usually the backslash character is used to escape them.

With this list of operators in hand, let's deconstruct our `find` command. When viewed from the uppermost level, we see that our tests are arranged as two groupings separated by an `-or` operator.

`( expression 1 ) -or ( expression 2 )`

This makes sense, since we are searching for files with a certain set of permissions and

for directories with a different set. If we are looking for both files and directories, why do we use `-or` instead of `-and`? As `find` scans through the files and directories, each one is evaluated to see whether it matches the specified tests. We want to know whether it is *either* a file with bad permissions *or* a directory with bad permissions. It can't be both at the same time. So if we expand the grouped expressions, we can see it this way:

```
( file with bad perms ) -or ( directory with bad perms )
```

Our next challenge is how to test for "bad permissions." How do we do that? Actually, we don't. What we will test for is "not good permissions" since we know what "good permissions" are. In the case of files, we define good as 0600 and for directories, we define it as 0700. The expression that will test files for "not good" permissions is as follows:

```
-type f -and -not -perms 0600
```

For directories it is as follows:

```
-type d -and -not -perms 0700
```

As noted in the Table 17-4 above, the `-and` operator can be safely removed since it is implied by default. So if we put this all back together, we get our final command.

```
find ~ ( -type f -not -perms 0600 ) -or ( -type d -not -
perms 0700 )
```

However, since the parentheses have special meaning to the shell, we must escape them to prevent the shell from trying to interpret them. Preceding each one with a backslash character does the trick.

There is another feature of logical operators that is important to understand. Let's say that we have two expressions separated by a logical operator.

```
expr1 -operator expr2
```

In all cases, *`expr1`* will always be performed; however, the operator will determine whether *`expr2`* is performed. Table 17-5 outlines how it works.

Table 17-5: `find` AND/OR Logic

Results of *`expr1`*	Operator	*`expr2`* is...
True	`-and`	Always performed
False	`-and`	Never performed
True	`-or`	Never performed
False	`-or`	Always performed

Why does this happen? It's done to improve performance. Take `-and`, for example. We

know that the expression *`expr1 -and expr2`* cannot be true if the result of *`expr1`* is false, so there is no point in performing *`expr2`*. Likewise, if we have the expression *`expr1 -or expr2`* and the result of *`expr1`* is true, there is no point in performing *`expr2`*, as we already know that the expression *`expr1 -or expr2`* is true.

OK, so it helps it go faster. Why is this important? It's important because we can rely on this behavior to control how actions are performed, as we will soon see.

Predefined Actions

Let's get some work done! Having a list of results from our `find` command is useful, but what we really want to do is act on the items on the list. Fortunately, `find` allows actions to be performed based on the search results. There are a set of predefined actions and several ways to apply user-defined actions. First, let's look at a few of the predefined actions listed in Table 17-6.

Table 17-6: Predefined `find` Actions

Action	Description
`-delete`	Delete the currently matching file.
`-ls`	Perform the equivalent of `ls -dils` on the matching file. Output is sent to standard output.
`-print`	Output the full pathname of the matching file to standard output. This is the default action if no other action is specified.
`-quit`	Quit once a match has been made.

As with the tests, there are many more actions. See the `find` man page for full details.

In the first example, we did this:

```
find ~
```

This produced a list of every file and subdirectory contained within our home directory. It produced a list because the `-print` action is implied if no other action is specified. Thus, our command could also be expressed as follows:

```
find ~ -print
```

We can use `find` to delete files that meet certain criteria. For example, to delete files that have the file extension `.bak` (which is often used to designate backup files), we could use this command:

```
find ~ -type f -name '*.bak' -delete
```

In this example, every file in the user's home directory (and its subdirectories) is searched for filenames ending in `.bak`. When they are found, they are deleted.

> **Warning:** It should go without saying that you should ***use extreme caution*** when using the `-delete` action. Always test the command first by substituting the `-print` action for `-delete` to confirm the search results.

Before we go on, let's take another look at how the logical operators affect actions. Consider the following command:

```
find ~ -type f -name '*.bak' -print
```

As we have seen, this command will look for every regular file (`-type f`) whose name ends with `.bak` (`-name '*.bak'`) and will output the relative pathname of each matching file to standard output (`-print`). However, the reason the command performs the way it does is determined by the logical relationships between each of the tests and actions. Remember, there is, by default, an implied `-and` relationship between each test and action. We could also express the command this way to make the logical relationships easier to see:

```
find ~ -type f -and -name '*.bak' -and -print
```

With our command fully expressed, let's look at how the logical operators affect its execution:

Test/Action	Is Performed Only If...
`-print`	`-type f` and `-name '*.bak'` are true
`-name '*.bak'`	`-type f` is true
`-type f`	Is always performed, since it is the first test/action in an `-and` relationship.

Since the logical relationship between the tests and actions determines which of them are performed, we can see that the order of the tests and actions is important. For instance, if we were to reorder the tests and actions so that the `-print` action was the first one, the command would behave much differently.

```
find ~ -print -and -type f -and -name '*.bak'
```

This version of the command will print each file (the `-print` action always evaluates to true) and then test for file type and the specified file extension.

User-Defined Actions

In addition to the predefined actions, we can also invoke arbitrary commands. The traditional way of doing this is with the `-exec` action. This action works like this:

`-exec` *`command`* `{} ;`

Here *command* is the name of a command, {} is a symbolic representation of the current pathname, and the semicolon is a required delimiter indicating the end of the command. Here's an example of using `-exec` to act like the `-delete` action discussed earlier:

```
-exec rm '{}' ';'
```

Again, since the brace and semicolon characters have special meaning to the shell, they must be quoted or escaped.

It's also possible to execute a user-defined action interactively. By using the `-ok` action in place of `-exec`, the user is prompted before execution of each specified command.

```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

In this example, we search for files with names starting with the string `foo` and execute the command `ls -l` each time one is found. Using the `-ok` action prompts the user before the `ls` command is executed.

Improving Efficiency

When the `-exec` action is used, it launches a new instance of the specified command each time a matching file is found. There are times when we might prefer to combine all of the search results and launch a single instance of the command. For example, rather than executing the commands like this:

`ls -l` *`file1`*

`ls -l` *`file2`*

we may prefer to execute them this way:

`ls -l` *`file1 file2`*

This causes the command to be executed only one time rather than multiple times. There are two ways we can do this: the traditional way, using the external command `xargs` and the alternate way, using a new feature in `find` itself. We'll talk about the alternate way first.

By changing the trailing semicolon character to a plus sign, we activate the ability of `find` to combine the results of the search into an argument list for a single execution of the desired command. Going back to our example, this will execute `ls` each time a matching file is found:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

By changing the command to the following:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' +
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

we get the same results, but the system has to execute the `ls` command only once.

xargs

The `xargs` command performs an interesting function. It accepts input from standard input and converts it into an argument list for a specified command. With our example, we would use it like this:

```
find ~ -type f -name 'foo*' -print | xargs ls -l
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

Here we see the output of the `find` command piped into `xargs`, which, in turn, constructs an argument list for the `ls` command and then executes it.

Note: While the number of arguments that can be placed into a command line is quite large, it's not unlimited. It is possible to create commands that are too long for the shell to accept. When a command line exceeds the maximum length supported by the system, `xargs` executes the specified command with the maximum number of arguments possible and then repeats this process until standard input is exhausted. To see the maximum size of the command line, execute `xargs` with the `--show-limits` option.

Dealing with Funny Filenames

Unix-like systems allow embedded spaces (and even newlines!) in filenames. This causes problems for programs like `xargs` that construct argument lists for other programs. An embedded space will be treated as a delimiter, and the resulting command will interpret each space-separated word as a separate argument. To overcome this, `find` and `xargs` allow the optional use of a *null character* as an argument separator. A null character is defined in ASCII as the character represented by the number zero (as opposed to, for example, the space character, which is defined in ASCII as the character represented by the number 32). The `find` command provides the action `-print0`, which produces null-separated output, and the `xargs` command has the `--null` (or `-0`) option, which accepts null separated input. Here's an example:

```
find ~ -iname '*.jpg' -print0 | xargs --null ls -l
```

Using this technique, we can ensure that all files, even those containing embedded spaces in their names, are handled correctly.

A Return to the Playground

It's time to put `find` to some (almost) practical use. We'll create a playground and try some of what we have learned.

First, let's create a playground with lots of subdirectories and files.

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100}
[me@linuxbox ~]$ touch playground/dir-{001..100}/file-{A..Z}
```

Marvel at the power of the command line! With these two lines, we created a playground directory containing 100 subdirectories each containing 26 empty files. Try that with the GUI!

The method we employed to accomplish this magic involved a familiar command (`mkdir`), an exotic shell expansion (braces), and a new command, `touch`. By combining `mkdir` with the `-p` option (which causes `mkdir` to create the parent directories of the specified paths) with brace expansion, we were able to create 100 subdirectories.

The `touch` command is usually used to set or update the access, change, and modify times of files. However, if a filename argument is that of a nonexistent file, an empty file is created.

In our playground, we created 100 instances of a file named `file-A`. Let's find them.

```
[me@linuxbox ~]$ find playground -type f -name 'file-A'
```

Note that unlike `ls`, `find` does not produce results in sorted order. Its order is determined by the layout of the storage device. We can confirm that we actually have 100 instances of the file this way.

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l
100
```

Next, let's look at finding files based on their modification times. This will be helpful when creating backups or organizing files in chronological order. To do this, we will first create a reference file against which we will compare modification time.

```
[me@linuxbox ~]$ touch playground/timestamp
```

This creates an empty file named `timestamp` and sets its modification time to the current time. We can verify this by using another handy command, `stat`, which is a kind of souped-up version of `ls`. The `stat` command reveals all that the system understands about a file and its attributes.

```
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0       IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2018-10-08 15:15:39.000000000 -0400
Modify: 2018-10-08 15:15:39.000000000 -0400
Change: 2018-10-08 15:15:39.000000000 -0400
```

If we use `touch` again and then examine the file with `stat`, we will see that the file's times have been updated.

```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0       IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2018-10-08 15:23:33.000000000 -0400
Modify: 2018-10-08 15:23:33.000000000 -0400
Change: 2018-10-08 15:23:33.000000000 -0400
```

Next, let's use `find` to update some of our playground files.

```
[me@linuxbox ~]$ find playground -type f -name 'file-B' -exec touch
'{}' ';'
```

This updates all files in the playground named `file-B`. Next we'll use `find` to identify the updated files by comparing all the files to the reference file `timestamp`.

```
[me@linuxbox ~]$ find playground -type f -newer playground/timestamp
```

The results contain all 100 instances of `file-B`. Since we performed a `touch` on all the files in the playground named `file-B` after we updated `timestamp`, they are now "newer" than `timestamp` and thus can be identified with the `-newer` test.

Finally, let's go back to the bad permissions test we performed earlier and apply it to `playground`.

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 \) -or \(
-type d -not -perm 0700 \)
```

This command lists all 100 directories and 2,600 files in `playground` (as well as `timestamp` and `playground` itself, for a total of 2,702) because none of them meets our definition of "good permissions." With our knowledge of operators and actions, we can add actions to this command to apply new permissions to the files and directories in our playground.

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod
0700 '{}' ';' \)
```

On a day-to-day basis, we might find it easier to issue two commands, one for the directories and one for the files, rather than this one large compound command, but it's nice to know that we can do it this way. The important point here is to understand how the operators and actions can be used together to perform useful tasks.

Options

Finally, we have the options. The options are used to control the scope of a `find` search. They may be included with other tests and actions when constructing `find` expressions. Table 17-7 lists the most commonly used `find` options.

Table 17-7: `find` Options

Option	Description
`-depth`	Direct `find` to process a directory's files before the directory itself. This option is automatically applied when the `-delete` action is specified.
`-maxdepth` *`levels`*	Set the maximum number of levels that `find` will descend into a directory tree when performing tests and actions.
`-mindepth` *`levels`*	Set the minimum number of levels that `find` will descend into a directory tree before applying tests and actions.
`-mount`	Direct `find` not to traverse directories that are mounted on other file systems.

`-noleaf`	Direct `find` not to optimize its search based on the assumption that it is searching a Unix-like file system. This is needed when scanning DOS/Windows file systems and CD-ROMs.

Summing Up

It's easy to see that `locate` is as simple as `find` is complicated. They both have their uses. Take the time to explore the many features of `find`. It can, with regular use, improve your understanding of Linux file system operations.

Further Reading

- The `locate`, `updatedb`, `find`, and `xargs` programs are all part the GNU Project's *findutils* package. The GNU Project provides a website with extensive on-line documentation, which is quite good and should be read if you are using these programs in high security environments:
  http://www.gnu.org/software/findutils/